MPWide Manual

Table of Contents

1. Introduction

This manual briefly describes how to install and use MPWide. Derek Groen wrote it, and you can contact him at djgroennl@gmail.com.

2. Compiling MPWide

2.1 Compiling MPWide on Unix-based systems

On most Unix-based systems, most of MPWide can be compiled simply by typing:

make

in the root MPWide directory. This command compiles the following binaries in the same directory:

MPWUnitTests – unit tests that can be run locally to verify the MPWide installation.
MPWTestConcurrent – functional test which tests blocking and non-blocking communications locally between two threads.
MPWTest – performance test which is started on two machines, and performs a ping-pong test.
MPWForwarder – MPWide message forwarding demon.
MPWFileCopy – low-level binary used by the mpw-cp script (it is not used directly).
MPWDataGather – largely untested client for synchronizing directories in real time.
libMPW.a, libMPW.so OR libMPW.dylib – The MPWide libraries to link against.

2.2 Compiling MPWide on Mac OSX-based systems

To install MPWide on MacOSX, you need to make one modification in the Makefile.

Change the section:

```
# OS X
#SO_EXT = dylib
#SHARED_LINK_FLAGS = -dynamiclib

# Linux
SO_EXT = so
SHARED_LINK_FLAGS = -shared
```

To:

```
# OS X
SO_EXT = dylib
SHARED_LINK_FLAGS = -dynamiclib

# Linux
#SO_EXT = so
#SHARED_LINK_FLAGS = -shared
```

Once you have done this, feel free to proceed with the installation instructions, as they are given above for any Unix-based system.

3 Testing the main message passing functionalities of MPWide

This MPWide distribution comes with three tests: **MPWUnitTests**, **MPWTestConcurrent** and **MPWTest**. MPWUnitTests and MPWTestConcurrent are intended to verify the correctness of your MPWide installation, while MPWTest is intended to allow performance and connection tests between multiple machines.

The unit tests are run by simply typing:

```
./MPWUnitTests
```

These tests verify a number of low-level functions, ensuring that they do not crash and return expected values. If run correctly, this test suite will give the following output at the end:

```
"Unit tests completed. Number of failed tests: 0
Please also run MPW_Functionaltests to more completely test MPWide."
```

The functional tests are run by typing:

```
./MPWTestConcurrent
```

If run correctly, this test suite will give the following output at the end:

```
"MPWTestConcurrent completed successfully."
```

3.1 Testing the performance of MPWide

You can also test the connectivity and performance between machines. Obviously, for this test you need to have two machines available, each of which has an MPWide installation of the same version. Once MPWide has been set up on both machines, you can do a performance test by typing:

First, on machine #1:

```
./MPWTest 1 0.0.0.0
```

And then, on machine #2:

```
./MPWTest 0 <ip address or hostname of machine #1>
```

MPWide will attempt to establish a connection and then try to transfer a message back and forth between the machines 20 times in total. By default MPWide uses a single stream and an 8kB message size, but these settings can be adjusted by changing the command examples above. In fact the full sytax is:

```
./MPWTest <act_as_server?> <hostname or ip address of other endpoint> <number of streams (default: 1)> <message size in kilobytes (default: 8 kB))>
```

We recommend using 1 stream for performance tests over a local area network, and at least 64 streams for tests over a wide area network. The message size can be set to any size that fits well into the machine's physical memory (we usually use message sizes of 1GB or less).

4 Using MPWide in your application

A documented example of how to use MPWide in a C++ code can be found in tests/Test.cpp. Here we will describe how to use several key functionalities in MPWide:

4.1 Including MPWide.h and linking to MPWide

The main interface file of MPWide is MPWide.h. You can include it by typing:

```
#include "MPWide.h"
```

provided that the MPWide main directory is in your include path. The libraries required to link your MPWide program are in the same directory, so make sure that you somehow include that directory in the library paths of your application (e.g., by using the compiler flag "-L" or by adding it to your LD_LIBRARY_PATH).

4.2 Initializing MPWide

In earlier versions of MPWide, users were required to use the MPW_Init() function. However, this is no longer necessary, and users can freely create and destroy paths without using MPW_Init().

4.3 Creating and destroying MPWide paths

MPWide paths are connections that MPWide aims to establish and use between different machines. Paths can also be connected to a different instance of MPWide on the same machine, and we usually do this for testing purposes. One path can consists of multiple *TCP streams*, to allow superior communication performance.

Creating path is best done in two steps. The first step is to create the data structure that holds the administrative information. This is done using the function *MPW_CreatePathWithoutConnect()*. This function returns an identifier for the path, and takes three arguments: a string containing the host name of the other end point (or "0.0.0.0" if it will act as a listening server connection), a base post number that it will use, and an integer which passes the number of TCP streams that MPWide needs to establish in this path. The full function signature is:

```
int MPW_CreatePathWithoutConnect(string host, int server_side_base_port, int
streams_in_path);
```

and an example, creating 4 streams starting from port 16256 which intends to connect to the machine my.test.ac.uk is:

```
int path_id = MPW_CreatePathWithoutConnect("my.test.ac.uk", 16256, 4);
```

The second step is to actually connect the path to the other machine, or to start listening for incoming connections in the case that the path is meant to act in a server role. This is done using *MPW_ConnectPath()*. This function has two argument, which is the identifier returned by *MPW_CreatePathWithoutConnect()*, and a boolean that should be set to 'true' if the function is to act as a server and listen to incoming connections when it is unable to connect as a client itself. The full function signature is:

```
int MPW_ConnectPath(int path_id, bool server_wait);
```

And an example of its use (relying on the previous declared and initialized variable path_id) is:

```
int status  = MPW_ConnectPath(path_id, true);
```

Paths can be easily destroyed by using *MPW_DestroyPath()*. This function requires only one argument, namely the identifier of the path to destroy. In our case, we would therefore use:

```
int status = MPW_DestroyPath(path_id);
```

4.4 Exchanging data using the MPWide paths.

The main modes of exchange in MPWide is through *MPWSend()*, *MPWRecv()* and *MPWSendRecv()*. These functions have the following signatures:

```
int MPW_Send(char* sendbuf, long long int sendsize, int path_id);
int MPW_Recv(char* recvbuf, long long int recvsize, int path_id);
int MPW_SendRecv(char* sendbuf, long long int sendsize, char* recvbuf, long long
int recvsize, int path_id);
```

Here, *sendbuf* is a character buffer of length *sendsize*, and *recvbuf* is a character buffer of length *recvsize*. Once more, *path_id* is the identifier of the path you wish to use; the one returned by *MPW_CreatePathWithoutConnect()*. All these functions are blocking, which means that they return once the send and receive operations have completed.

MPWide doesn't use any headers in its messages, so it is up to the user to ensure that the message size of the *MPW_Send()* call on one endpoint matches that of the *MPW_Recv()* call on the other endpoint.

4.4.1 Non-blocking communications

MPWide also supports once type of non-blocking communication, namely through the *MPW_ISendRecv()* call, which has the following signature:

```
int MPW_ISendRecv( char* sendbuf, long long int sendsize, char* recvbuf, long
long int recvsize, int path);
```

MPW_IsendRecv() works the same way as *MPW_SendRecv()* except that it returns as soon as the communications are initiated. Also, it returns a positive identifier number if successful, or a negative error code if it fails to start properly. The identifier can then be used to check whether the message exchange has completed, using:

```
bool MPW_Has_NBE_Finished(int NBE_id);
```

, which returns true if the communication has finished, or to wait until the message exchange has completed using:

```
void MPW_Wait(int NBE_id);
```

5 Forwarding messages with MPWide on intermediary nodes

The MPWForwarder is a program that provides tcp message forwarding using MPWide. It connects two port ranges from two different remote ip addresses, and does not proceed with forwarding until both ends of a given forwarding channel are connected.

Example execution:

```
./Forwarder < forward.cfg
```

Config file layout:

```
<address1 ip address>
<address1 baseport>
<address2 ip address>
<address2 baseport>
<number of streams>
(...repeat this for any other forwarding route...)
done
```

Example config file:

```
1.2.3.4
6000
5.6.7.8
7000
16
done
```

6 File transfers with MPWide

MPWide can be used to copy files directly from one location to another. To enable this, you need to install MPWide in the *$HOME/.mpwide* directory on each endpoint machine, and have a python interpreter available on both machines. You can then use the mpw-cp script to copy files and directories:

Here is an example syntax for a local cluster use with Gigabit Ethernet. We configure MPWide with 4 streams, pacing at 500 MB/s each and a tcp buffer of 256 kilobytes. Please note that the last two arguments are actually optional, and can be omitted if you prefer to use automatically tuned settings:

```
  ./mpw-cp machine-a:/home/you/yourfile
him@machine-b:/home/him/yourfileinhishome 4 500 256
```

The full syntax of the mpw-cpcommand is:

```
./mpw-cp <source host>:<source file or dir> <destination host>:<destination file
or dir> <number of streams> <optional: pacing rate in MB> <optional: tcp buffer
in kB>
```

By default MPWide will place the server on the destination host, but MPWide also supports a REVERSE mode, where the server is placed on the source host. This can help in bypassing tricky

firewalls. The full syntax for REVERSE mode is:

```
./mpw-cp s:<source host>:<source file or dir> <destination host>:<destination file or dir> <number of streams> <optional:pacing rate in MB> <optional:tcp buffer in kB>
```

6.1 Troubleshooting problems with mpw-cp

SYMPTOM: mpw-cp connects properly, but seems to hang, as no receiving file is created.

POSSIBLE CAUSE: This may not be an actual problem, as MPWide retains all data in memory until the transfer of the file is

completed. If the file is large, the network slow or MPWide poorly configured, it might take a while.

POSSIBLE SOLUTION 1: Wait for a few minutes, or try transferring a smaller file first to get an accurate picture of the achieved bandwidth.

SYMPTOM: mpw-cp hangs, the output mentions something like:

```
"[X] Trying to bind as server at <Y>. Result = 1"
```

POSSIBLE CAUSE: The firewall on the server side is blocking the incoming connections that MPWide tries to establish.

POSSIBLE SOLUTION 1: Run mpw-cp in REVERSE mode (or in normal mode if you are currently using REVERSE mode).

POSSIBLE SOLUTION 2: Ask your local administrator to open a range of ports in your firewall. Make sure the range is at least twice the number of streams you wish to use for your transfers.

POSSIBLE SOLUTION 3: Change the value of baseport on line 130 in mpw-cp.cpp and recompile on both endpoints. This will change the port range used by mpw-cp.cpp.

SYMPTOM: mpw-cp connects properly, but then hangs for a long time.

POSSIBLE CAUSE: Unknown, I have only encountered this once. Could be a platform-specific issue or a MPWide bug.

POSSIBLE SOLUTION 1: Run mpw-cp on the other end-point node.

7 Using the MPWide Python interface with Cython

Before the MPWide Python interface can be used, you will need to compile it using Cython. This can be done by running

```
python cythonize.py build_ext –inplace
```

in the cython/ subdirectory. Naturally, you will require the Cython module for this to work. Once that is done, you can run **MPWTest.py** in the same way as the normal **MPWTest** benchmark program. The MPWide Python interface supports only a subset of all functions (primarily *MPW_CreatePath()*, *MPW_DestroyPath()* and *MPW_SendRecv()*), but we intend to expand this in later versions.

We also provide a SWIG-based Python interface in the distribution, but we recommend the Cython-based interface, as it is more portable across different platforms.